Parallel and Distributed Systems - CUDA Ising Model Report

Spyridon Baltsas - AEM: 10443

1 Summary

This report is about my proposed solutions for impelementing the Ising Model in both CPU and GPU. To be more specific, here are briefly presented and explained the algorithms I have implemented for my approach, for both single CPU process and GPU parallelization use cases. Moreover, we will examine the overall performance, efficiency and scalability of all algorithms using the required charts and tables. The source code, building instructions and the usage of the produced binaries are available in *this repository* and its *README*.

2 The model

2.1 Introduction

The Ising Model is a statistical mechanics model for ferromagnetic materials. The model consists of discrete variables having only two possible values (-1,+1), representing the magnetic dipoles within the material. After finite time, it reaches an equilibrium with regions of positive and negative magnetic moments (spin). [1]

2.2 Simulation

Thanks to its discrete nature, we may simulate the mentioned model using a cellular automaton. This cellular automaton for its operation is using von-Neumann (cross) neighbouring [2], periodic, cyclic, boundary conditions and the following rule applied for each cell, let k be the iterations;

$$M_{k+1}[i][j] = sign(M_k[i,j] + M_k[i-1][j] + M_k[i+1][j] + M_k[i][j-1] + M_k[i][j+1]) \tag{2.1}$$

3 Approach

Please note that for all the following algorithms, the lattice matrix is represented using row-major order. That is, if we have a $n \times n$ matrix, the element a_{ij} can be accessed using the following formula, avoiding the complexity of using double pointers.

$$a_{ij} = M_{n\times n}[i][j] = M_{n\times n}[n \cdot i + j] \tag{3.1}$$

Last but not least, in order to reduce memory usage as much as possible, all lattice matrices contain 1 byte only integers.

3.1 Sequential

The sequential implementation is quite straightforward. First of all, two arrays are created, one for the current lattice state, and one for the next lattice state. Next, every element of next state lattice is calculated using the rule (2.1). Also a helper `temp` pointer is used to swap those two arrays on every iteration, in order the next state to be the current one and continue the calculations. The time complexity of this algorithm is $\mathcal{O}(n^2)$.

3.2 CUDA parallelism

3.2.1 V1

For this version, we load array to the GPU global memory,and gets splitted into one-dimensional blocks. Each block contains a number of threads, and each thread is responsible for calculating the next lattice value of their corresponding element (one-to-one thread - element relationship). Therefore, there is no need for a double loop for this calculation anymore. Row, column and ID of the element can be found from the following snippet;

```
size_t elementID = blockDim.x*blockIdx.x + threadIdx.x;
size_t row = elementID / n, column = elementID % n;
```

Afterwards, the logic is similar to sequential, but without the double loop, by directly reading from the GPU global memory.

3.2.2 V2

For this version, in order to prepare for the final version version, lattice matrix is splitted using smaller squares. As a result, this time, two-dimensional blocks are required, with 1 thread each. Also, for V2 and V3 block size is the number of rows of the sub-square.

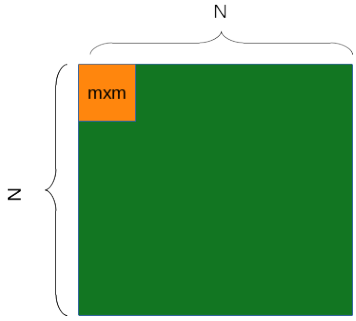

Figure 1: Splitting lattice matrix in smaller sub-squares

As a result, this time we need again a double loop to calculate each element of the sub-square. However, depending of the size of lattice matrix, may not fit perfectly in subsquares. The starting position of the loops and the required iteration which are depended on the size of lattice are calculated using the following snippet;

```
size_t blockRow = blockIdx.y*blockSize*n;
size_t blockCol = blockIdx.x*blockSize;
size_t rowIterations = n - blockRow/n < blockSize ? n-blockRow/n : blockSize;
size_t colIterations = n - blockCol < blockSize ? n- blockCol : blockSize;
```

Again, like on V2, for the calculations we read directly from global memory. However, this algorithm isn't as efficient, since we use only 1 thread per square and not taking the most of GPU, but still much faster than sequential.

3.2.3 V3

For this version, we use again squares like on V2 as shown on figure 1. This time, though, each element of the sub-square is assigned to a single thread, and calculations use the shared memory instead of the global. Thus, for this implementation we need again 2D blocks but with threads on both dimensions. With this implementation, like V1, there is no need for double loops, only checks whether we are within the limits of the lattice or not. The position of each element in the lattice matrix can be found by the following snippet;

```
size_t blockRow = blockIdx.y*blockDim.y;
size_t blockCol = blockIdx.x*blockDim.x;
size_t localRow = threadIdx.y, localCol = threadIdx.x, globalRow, globalCol;
globalCol = blockCol + localCol;
globalRow = blockRow + localRow;
```

1. Shared memory storage design

   In the shared memory 2D array, we must include all the elements of the sub-square, plus the neighboring elements of the sub-square. Also, for ease of calculations later, the elements of the subsquare must be in the middle, resulting in the following storage design. The elements in orange are the elements of the subsquare, and the elements in blue are the neighbors. In order to contain the neighbours of a $m \times m$ square, a $(m + 2) \times (m + 2)$ square is needed.

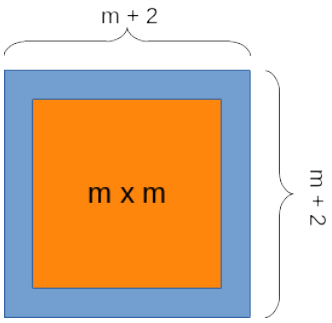

Figure 2: Shared memory array.

4 Test specifications

Please note that for the production of the following times, I/O operations like loading data to RAM or GPU memory is ommited. For the GPU runs, Aristotelis-HPC (Aristotle University High-Performance Computing infrastructure) was used. To be exact, the CUDA was run to a NVIDIA Tesla P100 (12 GB VRAM) [3]. For the sequential runs, an Intel Core i5-8300H @ 2.30 GHz (4 cores, 8 threads) was used.

5 Results

In the following results, N are the rows of square lattice and k the iterations. In addition, V2 and V3 were tested for block size equal to 16. In order to get maximum performance for each N, block size must be fine tuned by picking a value from 1 to 32, since 1024 threads are available for each block. For more detailed times of CUDA, please check the tables section.

5.1 Charts

5.1.1 N variable, k constant

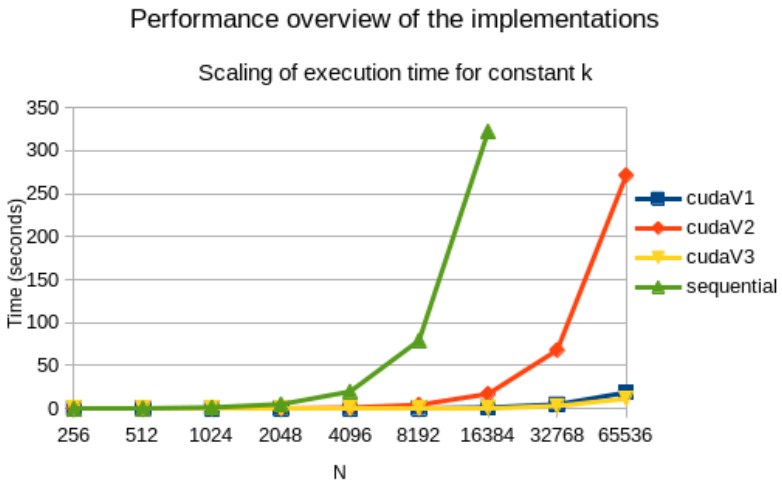

Figure 3: Performance of sequential and CUDA implementations for k = 50

5.1.2 k variable, N constant

1. Sequential

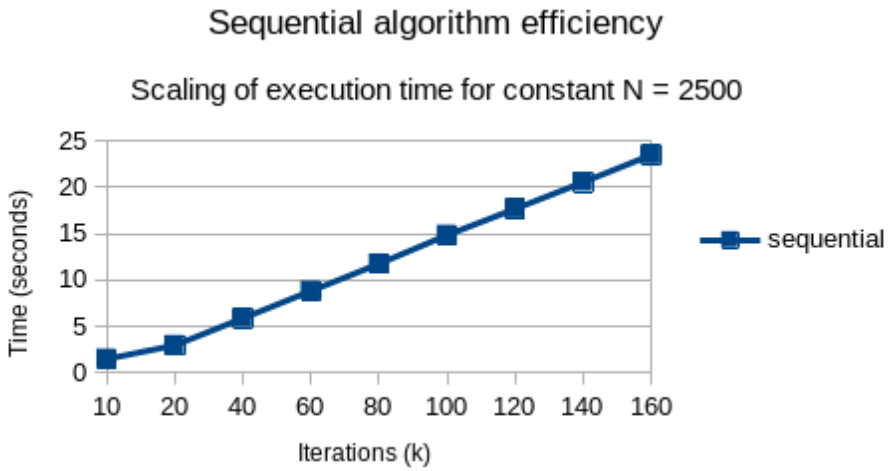

Figure 4: Scaling efficiency for sequential algorithm, N = 2500

2. CUDA

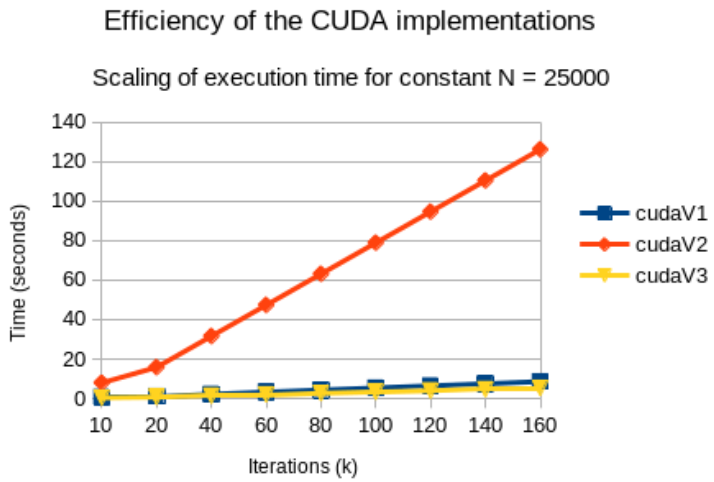

Figure 5: Scaling efficiency of CUDA algorithms, N = 25000

5.1.3 V2 optimal block size

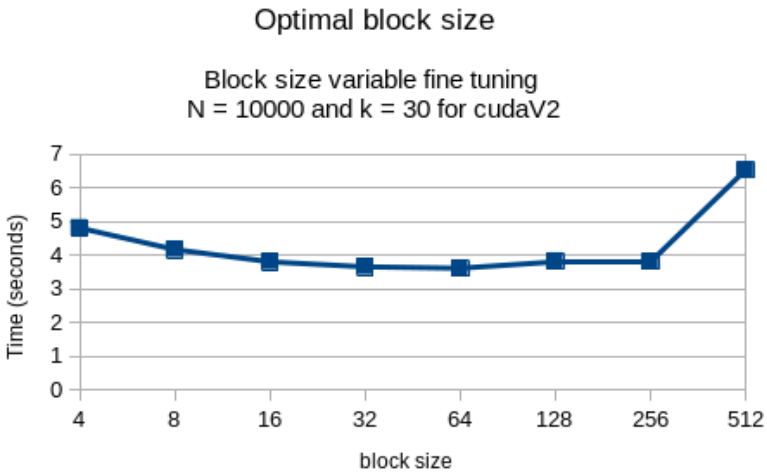

Figure 6: Optimizing block size

5.2 CUDA Tables

Table 1: Performance of CUDA algorithms for increasing N and k = 50

N	cudaV1	cudaV2	cudaV3
256	0.000792	0.012446	0.000418
512	0.001701	0.021732	0.001039
1024	0.005301	0.081199	0.00348
2048	0.019266	0.291012	0.01173
4096	0.074887	1.067174	0.0454
8192	0.294068	4.245557	0.179794
16384	1.166208	16.963191	0.717332
32768	4.649675	67.834809	2.878595
65536	18.582241	271.312043	11.475189

Table 2: Scaling efficiency of CUDA algorithms for increasing k and N = 25000

k	cudaV1	cudaV2	cudaV3
10	0.583251	7.953822	0.373243
20	1.114602	15.841907	0.697427
40	2.177255	31.601757	1.346061
60	3.239398	47.365691	2.006996
80	4.301688	63.137752	2.658596
100	5.365758	78.914083	3.302425
120	6.426835	94.682181	3.951027
140	7.488585	110.47514	4.601881
160	8.553132	126.247329	5.256477

6 References

[1] *Ising model — Wikipedia, the free encyclopedia*, `https://en.wikipedia.org/w/index.php?title=Ising_model&oldid=1191997935`, 2023.

[2] *Cellular automaton — Wikipedia, the free encyclopedia*, `https://en.wikipedia.org/w/index.php?title=Cellular_automaton&oldid=1196338105`, 2024.

[3] Κέντρο Ηλεκτρονικής Διακυβέρνησης, *Διαθέσιμοι Υπολογιστικοί πόροι*, `https://hpc.it.auth.gr/nodes-summary/`, 2023.